System Design Document for the Android application DrinkIT

Kajsa Bjäräng, Viktoria Enderstein, Elin Eriksson, Lisa Fahlbeck, Alice Olsson

2018-10-28
Version 7

Table of Contents

1 Introduction

DrinkIT is a pre party game application designed for groups of two to ten players. The game allows players to interact with each other and the games purpose is to create fellowship for new as well as old friends.

1.1 Design goals

The goal for the design of the application is to create a functional game with few dependencies that is easy to adapt and develop due to loose coupling. All classes should, to the greatest possible extent, have only one responsibility and not interfere with classes they have nothing to do with. The classes should be isolated and encapsulated.

1.2 Definitions, acronyms, and abbreviations

DrinkIT - The name of the application as well as the name of the main class in the model package.
Challenge - A task to perform for each player when it is their turn, for example answer a question or perform a song or charade.
Category - All challenges are divided into one of nine different categories.
GameRound - GameRound contains all information needed to distinguish and save statistics about any round during the game. It contains a player, a challenge, whether the challenge succeeded or failed, and a static list of GameRounds of the played rounds.
Gradle - An open-source build automation tool designed for multi-project builds. Gradle comes included in Android Studio.
Travis - Automatically builds the code and checks if everything still works after a push to GitHub. Otherwise it reports that something went wrong, which is good when several people work on the same project.
Android Studio - An integrated development environment (IDE). It's the official IDE for Google's Android Operating system.
JSON (JavaScript Object Notation) - A lightweight data-interchange format, which is both easy to write and to parse and generate. Based on a subset of JavaScript, but completely language independent.
Activity - In Android Studio, an Activity is a class with a connected xml file handling views.

2 System architecture

The application is made in Android Studio version 3.1.4, written in the language Java. It uses the built in automation tool Gradle version 4.4 which comes included in this version of Android Studio. The application is designed to be played on a single unit of an android smartphone. DrinkIT targets API 27 and Android 8.1, and supports all newer versions of OS.

2.1 Game flow

To start the game the user has to add all participating players by name, then choose which categories should be included, and then decide the duration of the game, which is calculated to a specific number of rounds. Thereafter a challenge from one of the chosen categories is randomly shown on the screen. Some challenges give the player points if they are accomplished whilst others are just for fun. After the number of challenges corresponding to the chosen duration have been played the game is finished and a scoreboard is shown with an option to continue playing. During the game there are options always available to add another player, remove an existing player, quit the game, or access instructions and help.

3 System design

The application is designed to make it easy for future developers to understand, navigate and expand the code. To achieve this the development of the application is built on a number of design principles and patterns.

3.1.1 Single responsibility principle

Classes doing one thing only, and doing it well, leads to a robust code. In the application DrinkIT one of the goals was to make sure all of the classes and class methods follows the Single Responsibility Principle. For example most classes in the model follow the Single Responsibility Principle, however there are still exceptions at this point in time, and this is a future improvement to work on. DrinkIT is an example of a class that does not follow the Single Responsibility Principle, and the refactoring of removing functionality from this class has been started but not completed. The remaining classes in the Model follow the principle, as they only contain functionality regarding themselves.

3.1.2 Open Closed Principle

Making the application open for extension but closed for modification is achieved by the implementation of design patterns. The Model-View-Controller pattern makes it easy for future developers to expand DrinkIT, since the Controller class handles all communication between different packages and therefore makes it easy to add more components or packages. The implementation of JSON-files also makes it easy to add more challenges by only having to add them once.

To make the application closed for modification, encapsulation has been used. The goal of encapsulation is to avoid global access of variables and methods which prevents data to be modified by an outside user. All variables and methods that do not need to be accessed outside of their classes are private, so they cannot be accessed and modified elsewhere in the code. This is an example of how the Open-Closed Principle is achieved in the application.

3.1.4 Model-View-Controller Pattern (MVC)

The goal of the dependency between classes is to achieve as high cohesion and as low coupling as possible. The MVC pattern helps keeping the coupling low by handling the communication between packages. Due to DrinkITs many views and game logic the implementation of the MVC pattern was a suitable choice.

The model of the application lies in the package Model where the class DrinkIT is the heart of the model. The View contains multiple Activities which lie in the View package and the Controller class lies in the Controller package and acts as a connection between the Model, View and Services packages.

The class ChallengePageOne gets information from the model regarding which is the current category, and then creates the view dynamically depending on that category. For example the category Quiz contains a question and an answer, while the category Rules only contains a challenge.

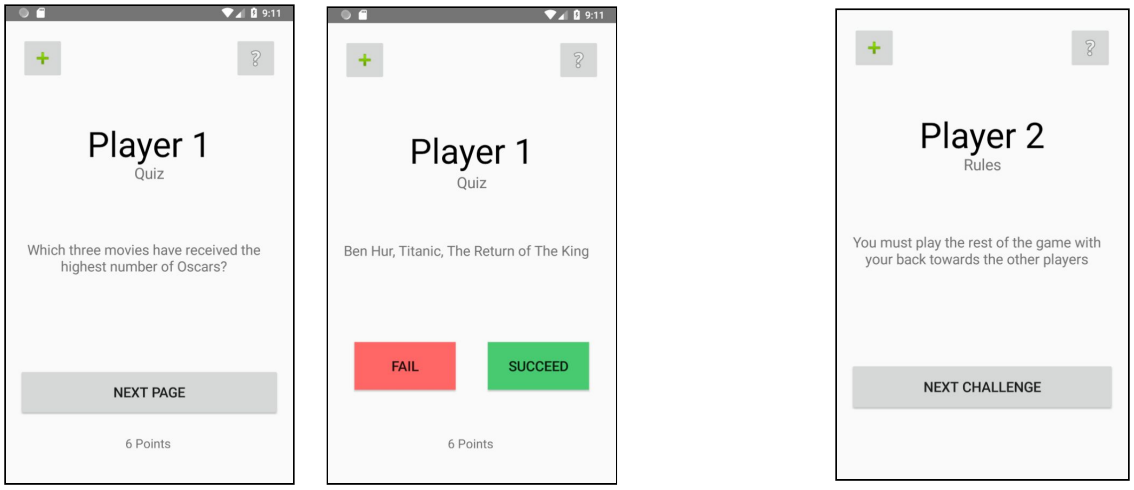

Figure 1. View of a challenge from the category “Quiz” with two pages vs a view of a challenge from the category “Rules”.

The model is responsible for the domain logic and holds several classes with different responsibilities. The model consists of five classes, Category, Challenge, DrinkIT, GameRound and Player. They all hold different functionality, but collaborate as a unit with few dependencies.

The class DrinkIT acts as an interface to the Model package from the Controller. It’s the Controller which interacts with the players, through input from the different views, and calls for the right methods from the model.

3.2 Design model

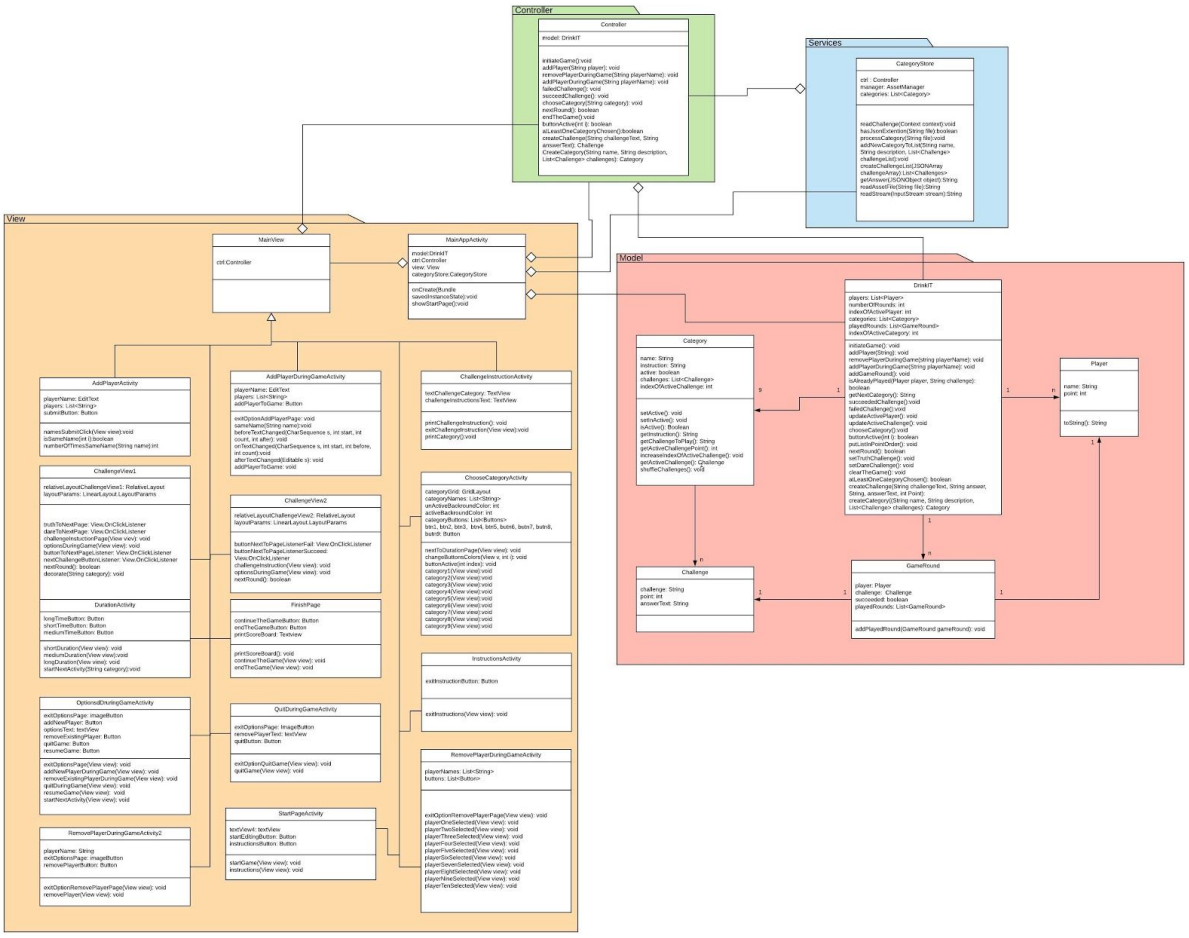

Figure 2. An overview of the whole project with all packages. For a more detailed view, see Appendix.

The design model consists of four different packages, Services, View, Model and Controller and follows, as earlier mentioned, the design pattern MVC. The class DrinkIT which lies in the Model package holds a key position. You could say that the DrinkIT class is at the heart of the game since it holds most of the underlying game logic.

MainAppActivity initializes the game and creates an instance of each main class - Controller, MainView and the model DrinkIT - which are used throughout the game. MainView is created to decrease dependencies by acting as a superclass to all other views, thereby holding the only instance of the Controller available to all Views.

When starting the game the startpage is a simple view with two buttons acting as a clear entry point to the game. When starting the game the user gets to add all players by manually entering their names. This view is designed so that the user dynamically receives error messages with the help of the Observer Pattern, found in AddPlayerActivity, if the input

entered is incorrect. The game is configured to accept a minimum of two players and no duplicate names.

DrinkIT contains a list of players. The class Player contains a name and a point, acting as a score. The players are added in the list in the beginning of the game as explained above. This list contains the players during the game and is used in different methods in DrinkIT.

DrinkIT also contains a list of categories. The class Category contains a name, an instruction on how to play the Category in question, a list of challenges, an index of the active challenge and a state, either active or inactive.

The user gets to choose from nine categories which they can include in the game. When selecting a category, its state gets set to active. DrinkIT contains an integer indexOfActiveCategory which is incremented after each game round to supply the user with the next challenge. Further on in the text, under *4. Persistent data management*, there can be found an explanation of how challenges are created.

GameRound is the class that assists the application in keeping track of all statistics regarding earlier played rounds. After every played round the player, the challenge and the outcome - if the played succeeded or failed - is stored in a list of game rounds i the GameRound class. DrinkIT uses the list to make sure the same player never has to play the same challenge twice.

To loop through the players and ensure each player gets to play the same number of rounds during the game, DrinkIT contains an integer, indexOfActivePlayer, which is incremented after each completed round. Every time the index reaches the last player in the list, the list is shuffled and the index is set back to 0. This is to avoid players consistently showing up in the same order, which makes the game more interesting and unexpected to play. There is also logic to prevent the same player showing up twice in a row, even as the list is shuffled and the index reset.

3.3 The test package

During the implementation of the application, tests have been implemented after every new implemented user story, to make sure the code that has been written is solid and works correctly. The application contains two test packages, com.TDA367.drinkit (test) and com.TDA367.drinkit (androidTest), which contains multiple unit tests, made with the framework JUnit. There is a 100% coverage of test on the classes but not on the methods when we didn't test get or set methods.

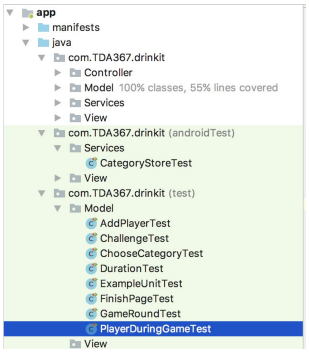

Figure 3. Where to find the tests

100% classes, 55% lines covered in package 'com.TDA367.drinkit.Model'

Element	Class, %	Method, %	Line, %
Category	100% (1/1)	75% (12/16)	76% (35/46)
Challenge	100% (1/1)	80% (4/5)	78% (11/14)
DrinkIT	100% (1/1)	63% (28/44)	49% (132/268)
GameRound	100% (1/1)	77% (7/9)	54% (17/31)
Player	100% (1/1)	80% (4/5)	75% (12/16)

Figure 4. Coverage of the tests

3.4 Dependency analysis

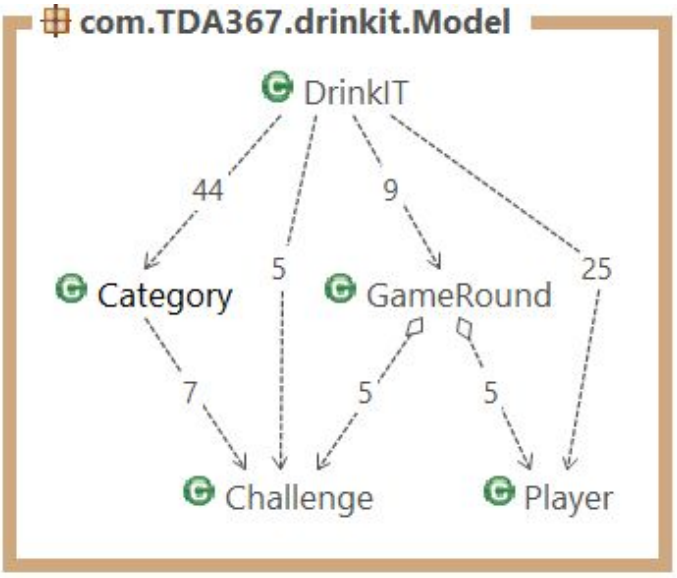

Figure 5. Shows the dependencies in the model package

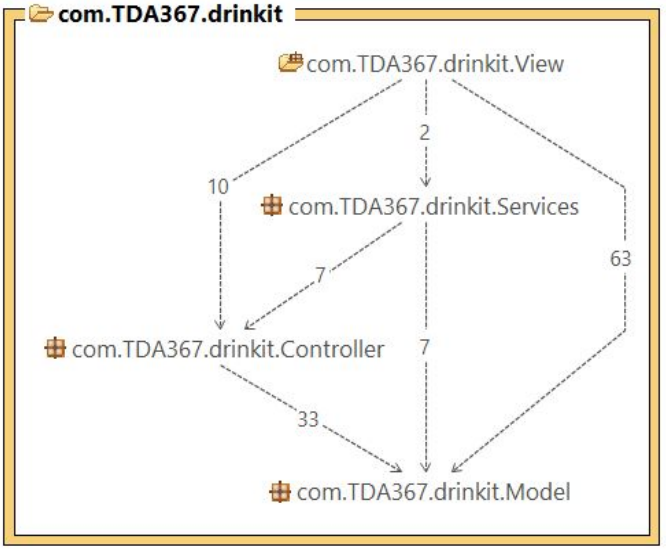

Figure 6. Shows the dependencies between the four packages model, view, controller and services.

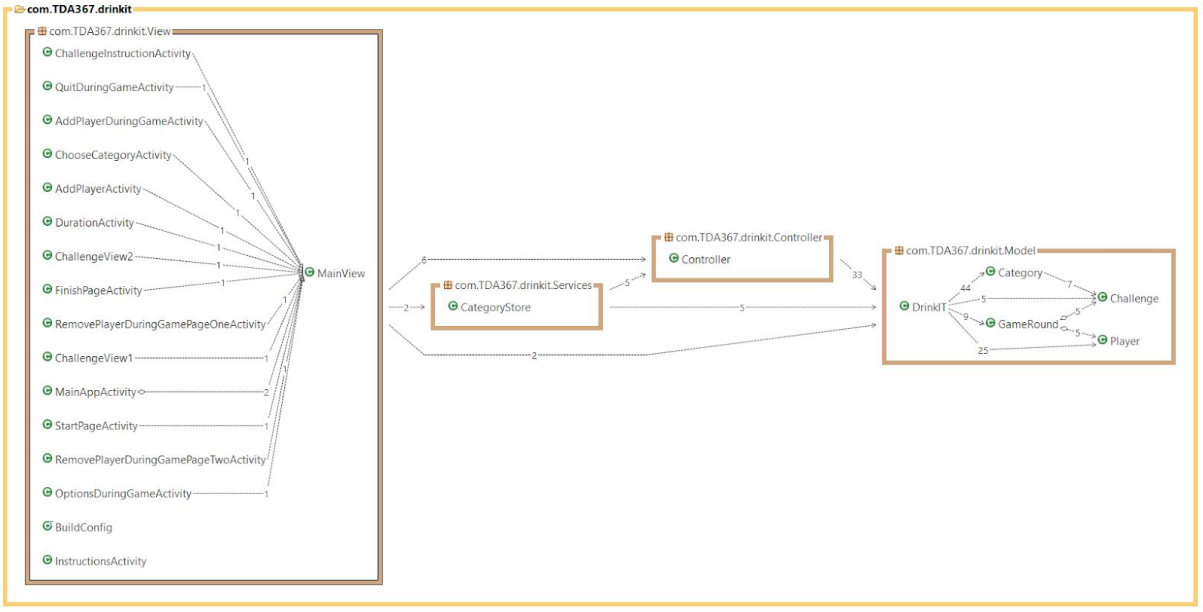

Figure 7. Shows all the dependencies in the project.

4 Persistent data management

Categories are created through files written in JSON format. They are parsed through, and the data in each file is used to generate the categories in the game and all challenges contained within them. Each file contains a name and a description of the category, as well as an array of challenges belonging to the category in question. Each challenge is made up of a challenge text and a point, with some also containing answers to the challenges posed.

When the application is initialized the .json files are parsed through in the class CategoryStore, situated in the package Services. All categories are collected in a list in the model DrinkIT, and its associated challenges are contained in lists in the categories.

This manner of creating categories and challenges lends itself well to further extension of the code. It is easy to add both new categories and new challenges, as the files are parsed through and generated in the same way independent of its size.

5 Potential improvement areas in code management

Like all applications, DrinkIT has both flaws and a potential for further development. There is a lot of potential for future developers concerning this application. Functionality can be added in some fields, for example, regarding the View and some components in the model.

One example of improvement regarding the code concern the class *AddPlayerActivity*. As of now an observer is implemented that checks whether two players have the same name. While the user writes in the name of the players the user gets direct feedback via an error message besides the textfield. The observer however does not work as planned due to the delay of the error message. Only when the user presses “delete” does the error message appear if the same name is already written in another textfield. This is an error which future developers should look over.

Another example concerns the class AddPlayerActivity and ChooseCategoryActivity which consist of a lot of duplicate code. At this moment in time, there is no easy way to add new categories to DrinkIT due to the nine hard coded categories in ChooseCategoryActivity. However this could be arranged differently and made more abstract by implementing methods which gets data from the model instead of just having rows of instantiations.

In FinishPageActivity there is a method, continueTheGame, which is not yet completed. The method should save all the input from the users but as of now it does not.

There is one place in the code where the categories are compared to Strings, which is when loading different views depending on the active category. This is necessary, but it is a good practice to keep such comparisons to the minimum, as it decreases the amount of code that would have to be edited when for example potentially adding a completely new category. It also violates the Open-Closed Principle since Strings are easy to modify.

Lastly, functionality regarding challenges that involves all players should not be directed to a single player. At this time, due to the connection between the list of players and list of categories that is not possible. This could be refactored by future developers.

6 References

None.

7 Appendix 1

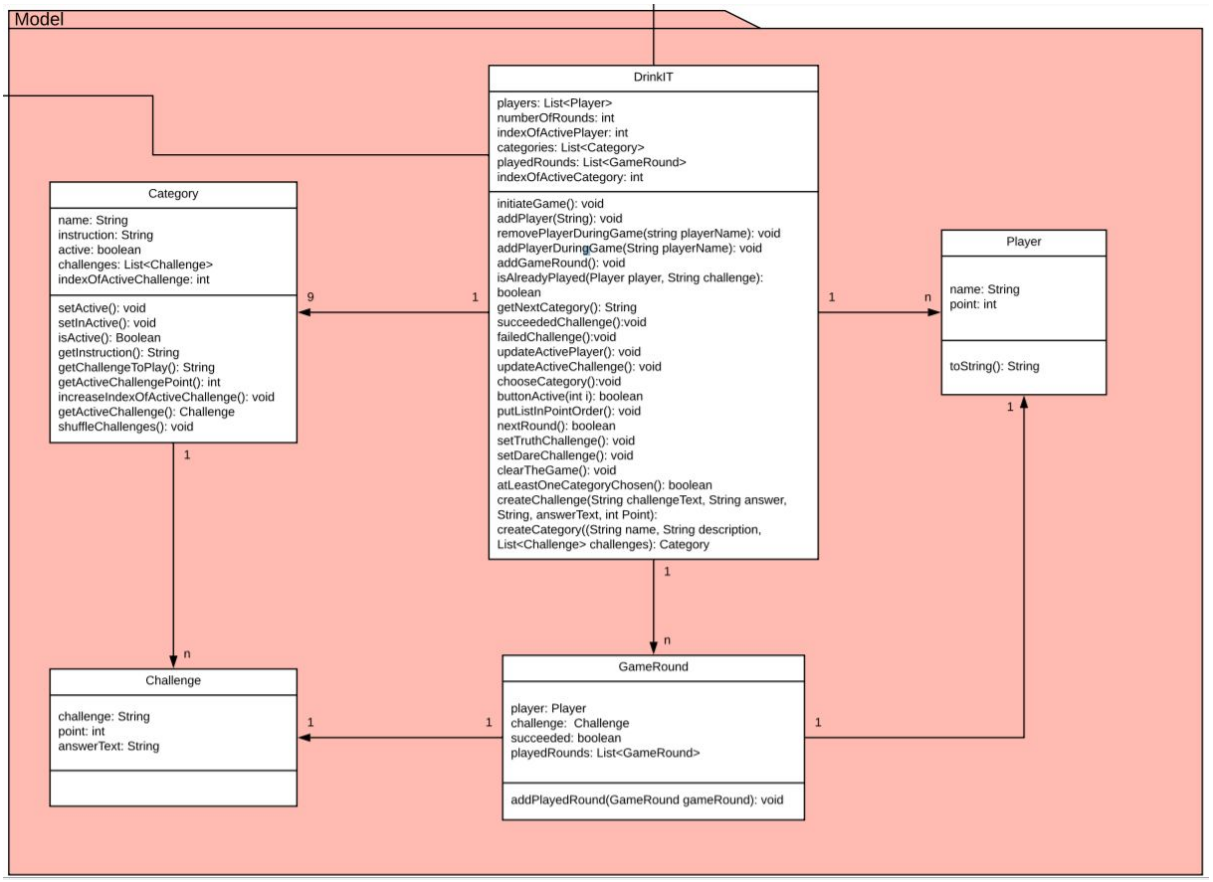

Figure 8: Shows the model package

8 Appendix 2

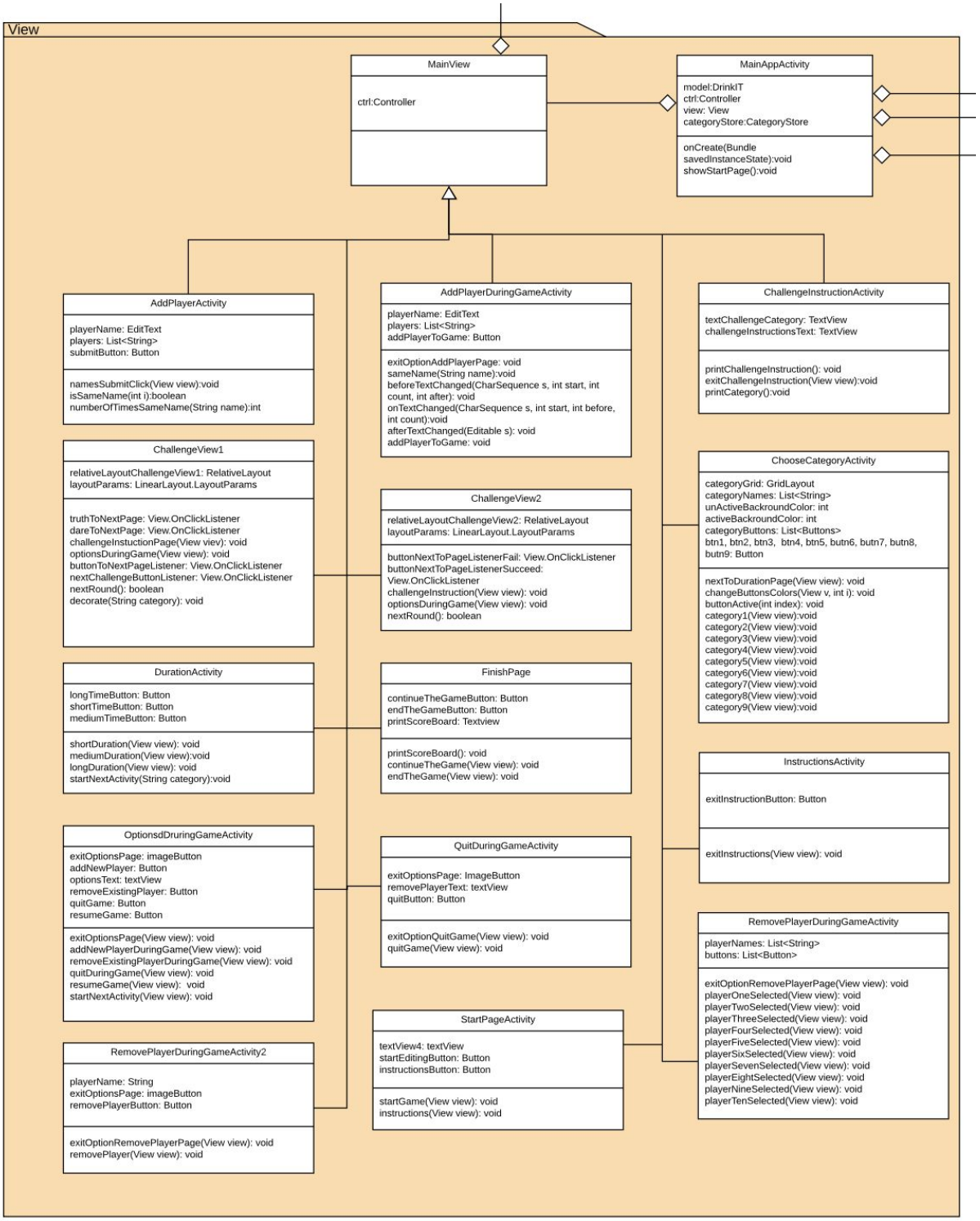

Figure 9: Shows the View package

9 Appendix 3

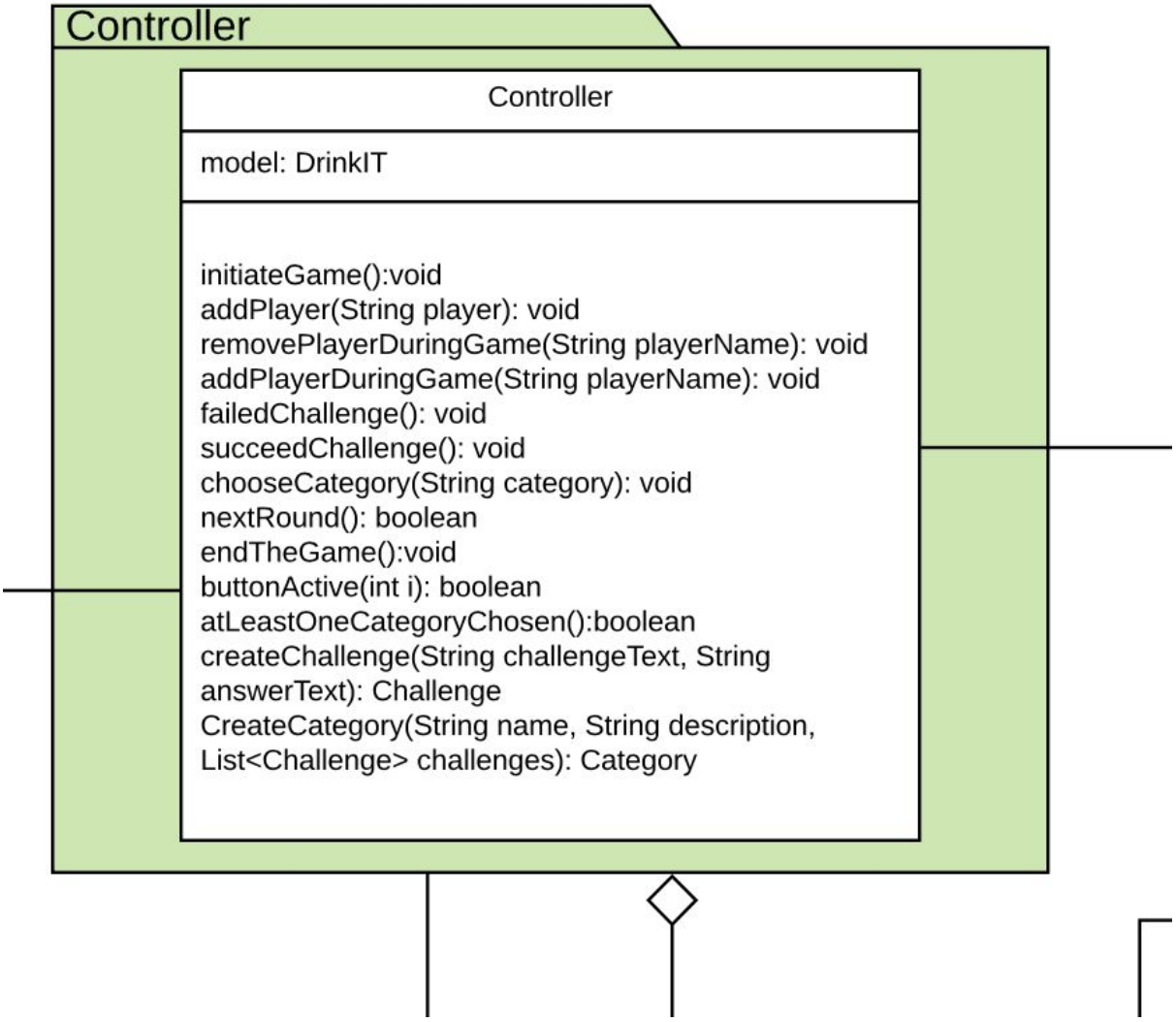

Figure 10: Shows the Controller package